Exploring Qualcomm Baseband via ModKit

Tencent Blade Team

Tencent Security Platform Department

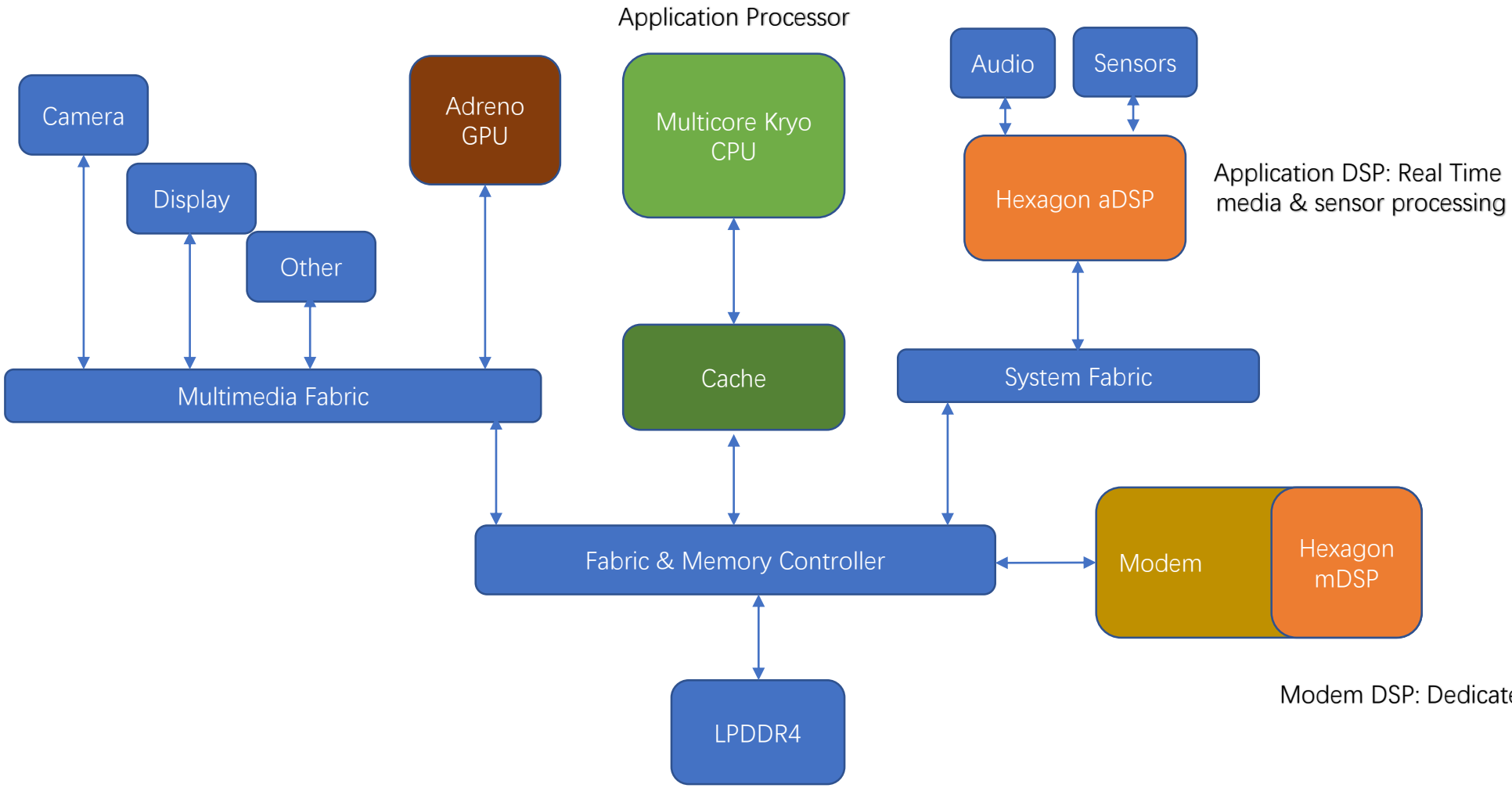
About Us - Tencent Blade Team

- A security research team from Tencent Security Platform Department
- Focus security research on AI, IoT and Mobile
- Have discovered 70+ security vulnerabilities
- Research output has been widely used in Tencent products
- Contact us: blade@tencent.com

Agenda

- **Enter Qualcomm Modem World**
- Static Analysis of Modem
- Debugging Modem with ModKit
- LTE Attack Surface Introduction

Google Pixel – MSM 8996 Pro



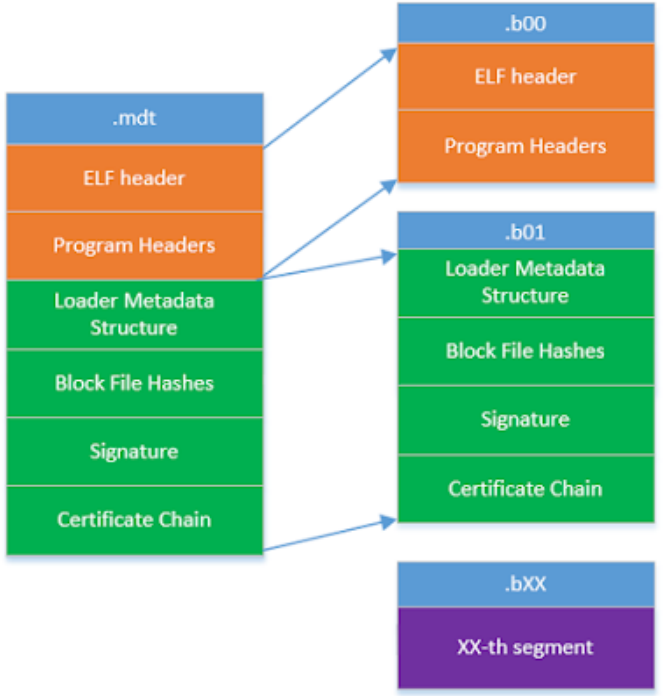
Hexagon DSP Processor

- Memory
  - Program code and data are stored in a unified 32-bit address space
  - little-endian
- Registers
  - 32 32-bit general purpose registers can be accessed as single registers or as 64-bit register pairs
- Parallel Execution
  - Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution
  - Each packet contains from 1 to 4 instructions
- Cache Memory
  - Separate L1 instruction and data caches exist for program code and data
  - Unified L2 cache
- Virtual Memory
  - Real-Time OS (**QuRT**) handles the virtual-to-physical memory mapping
  - Virtual Memory supports the memory management and protection

Modem Images (Google Pixel)

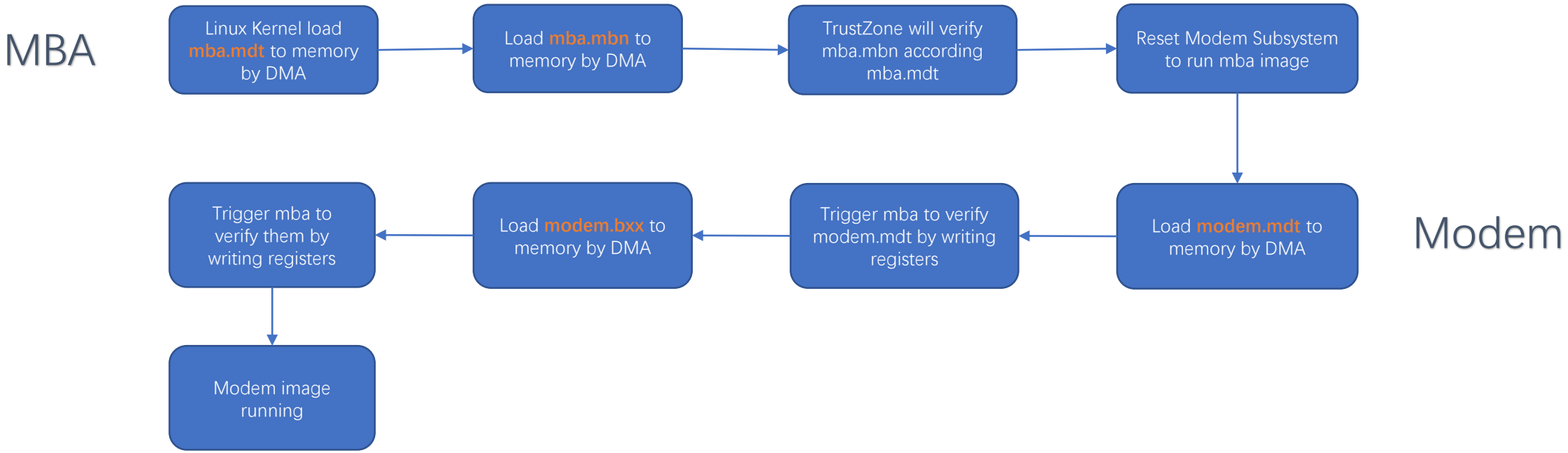
- Subsystem images formats according [laginimaineb's blog]
  - *.mdt: contains headers and information used to verify *.bxx
  - *.bxx: b00 contains headers, b01 contains verification information, others are segments
  - mba.mdt: MBA(Modem Boot Authenticator) image metadata
  - mba.mbn: MBA image file, a replacement of mba.bxx
  - modem.mdt: Modem image metadata
  - modem.bxx: Modem image files

```
sailfish:/firmware/radio # ls -l
total 55824
-r--r----- 1 system root      244 1980-01-01 00:00 mba.b00
-r--r----- 1 system root     4584 1980-01-01 00:00 mba.b01
-r--r----- 1 system root   153968 1980-01-01 00:00 mba.b02
-r--r----- 1 system root    10144 1980-01-01 00:00 mba.b03
-r--r----- 1 system root    31588 1980-01-01 00:00 mba.b04
-r--r----- 1 system root      896 1980-01-01 00:00 mba.b05
-r--r----- 1 system root   213888 1979-12-31 19:00 mba.mbn
-r--r----- 1 system root     4828 1980-01-01 00:00 mba.mdt
-r--r----- 1 system root      884 1980-01-01 00:00 modem.b00
-r--r----- 1 system root     5224 1980-01-01 00:00 modem.b01
-r--r----- 1 system root     5460 1979-12-31 19:00 modem.b02
-r--r----- 1 system root   196608 1979-12-31 19:00 modem.b03
-r--r----- 1 system root  2816788 1979-12-31 19:00 modem.b04
-r--r----- 1 system root  3288171 1979-12-31 19:00 modem.b05
-r--r----- 1 system root   163280 1979-12-31 19:00 modem.b06
-r--r----- 1 system root   735936 1979-12-31 19:00 modem.b07
-r--r----- 1 system root  2081092 1979-12-31 19:00 modem.b08
-r--r----- 1 system root 15348944 1979-12-31 19:00 modem.b09
-r--r----- 1 system root   343648 1979-12-31 19:00 modem.b10
-r--r----- 1 system root   488448 1979-12-31 19:00 modem.b11
-r--r----- 1 system root 11529784 1979-12-31 19:00 modem.b12
-r--r----- 1 system root  7234272 1979-12-31 19:00 modem.b13
-r--r----- 1 system root    82368 1979-12-31 19:00 modem.b15
-r--r----- 1 system root   530566 1979-12-31 19:00 modem.b16
-r--r----- 1 system root  9789440 1979-12-31 19:00 modem.b17
-r--r----- 1 system root    77824 1979-12-31 19:00 modem.b18
-r--r----- 1 system root  1323008 1979-12-31 19:00 modem.b19
-r--r----- 1 system root   408484 1979-12-31 19:00 modem.b20
-r--r----- 1 system root     6108 1979-12-31 19:00 modem.mdt
```



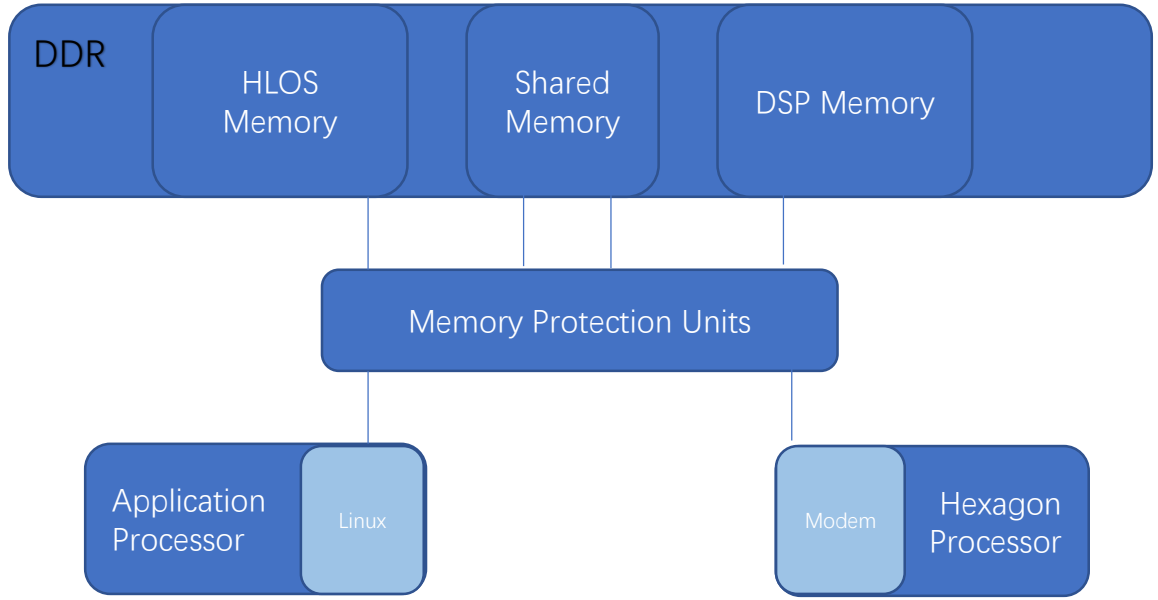
Modem Booting Process (Google Pixel)

- Linux Kernel is responsible for loading modem images to physical memory
- The Modem booting process on Google Pixel is as below graph.
- Linux kernel function **pil_boot** describes this process.


MBA
Linux Kernel load mba.mdt to memory by DMA
Load mba.mbn to memory by DMA
TrustZone will verify mba.mbn according mba.mdt
Reset Modem Subsystem to run mba image
Load modem.mdt to memory by DMA
Trigger mba to verify modem.mdt by writing registers
Load modem.bxx to memory by DMA
Trigger mba to verify them by writing registers
Modem image running
Modem

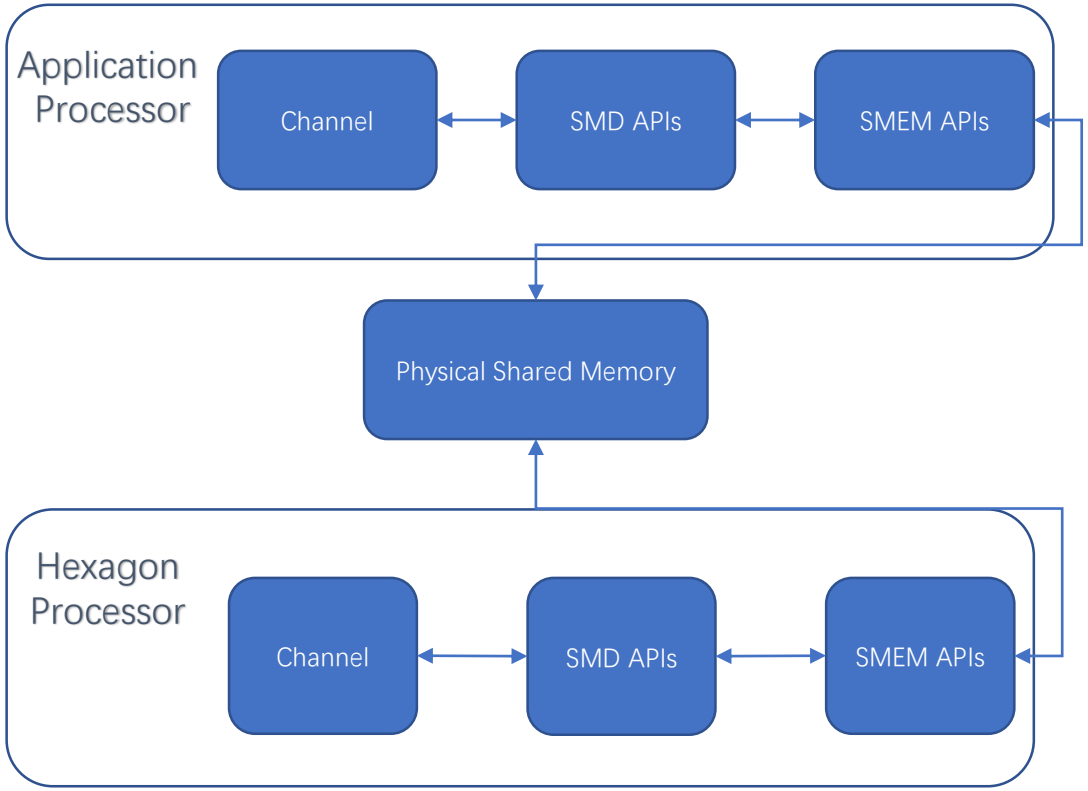
Communication Between Linux and Modem

- Modem is running on Hexagon Processor, communicates with Application Processor via **SMEM** (Shared Memory)
- Common SMEM APIs like smem_init / smem_alloc used in both Modem and Linux
- On Google Pixel, The Physical base of SMEM is 0x86000000, size is 0x200000



Communication Between Linux and Modem

- SMD (Shared Memory Driver, smd.c)
  - A wrapper of SMEM for data communication
  - There is a abstract object called smd_channel which is like a duplex pipe

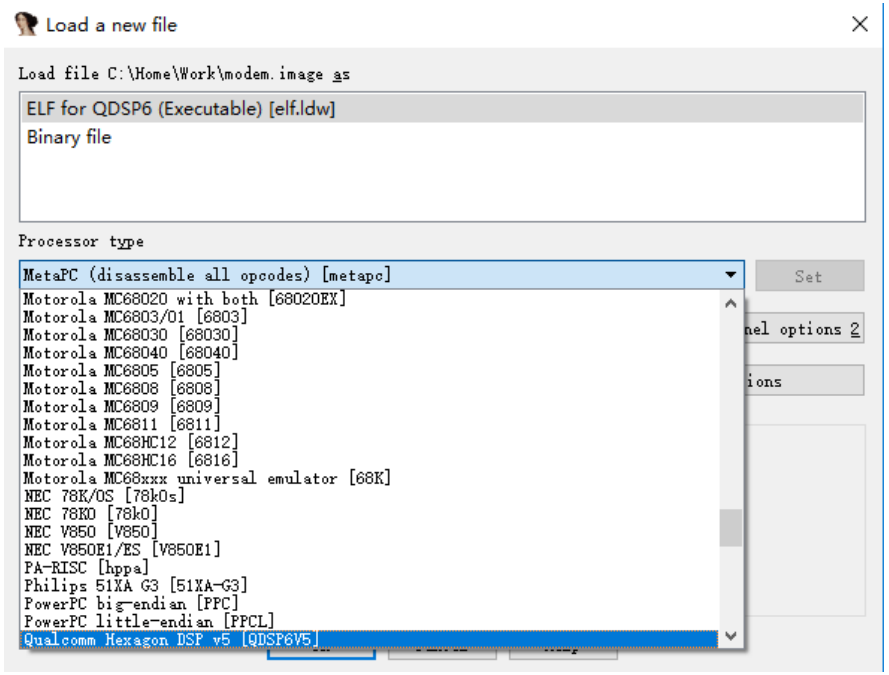


Agenda

- Enter Qualcomm Modem World
- **Static Analysis of Modem**
- Debugging Modem with ModKit
- LTE Attack Surface Introduction

Load Modem Images In IDA Pro

- Construct modem.bxx to a valid ELF file:
  - Read program headers from modem.mdt
  - Construct modem.bxx according to program headers
  - laginimaineb's python script
  - IDA Pro Processor Module for Hexagon: Hexag00n



Source Code

- Old version source code of MSM 8916 can be found on Internet, We can learn:
  - Modem network connection flow
  - OTA data handling flow
  - QuRT implementation, such as heap management
  - Many log strings are the same as MSM 8996 Pro on Google Pixel
  - A file called msg_hash.txt in the source catches our attention

Connect Binary Code to Log String

- msg_hash.txt
  - Split log strings from binary to reduce firmware size, save them in msg_hash.txt
  - Msg_hash.txt format: unique id + source file name + log string
  - Unique id = lower 4 bytes of md5(source file name + log string)
  - Useful in lastest firmware even if you only have a old Qshrink file
  - Contains rich information for RE
- In binary
  - Too many log functions
  - *(R0 + 4) = unique id
  - Compare *(R0 + 4) to md5 hash after located pattern of call log function

```
unknown@unknown-desktop:~$ cat msg_hash.txt | tail -n 10
6475835:qfe2340v3p0_asm.cpp:lte_dlca_rfm_vote = %d
1063147346:qfe2340v3p0_asm.cpp:ASM Disable_rx cannot be executed as there are active DLCA LTE bands in the ASM
98705263:qfe2340v3p0_asm.cpp:lte_ulca_rfm_vote = %d
2987490609:qfe2340v3p0_asm.cpp:ASM Disable_tx cannot be executed as there are active ULCA LTE bands in the ASM
1752231959:diag_mode.c:Buffer API status %d
293814359:diagcomm_io_udp.c:qpoll returned %d error
1139581605:qmi_voice_cm_if.c:FEATURE_ECALL_APP not defined. Cannot update eCall MSD setting
3536580398:cmapi_modem_data_get.c:  Sys Mode: %d | Area Code: %d | Global Cell Id: %d |  PSC_PID: %d | ARFCN:
245285565:cmapi_modem_data_get.c:    RSSI(P): %d | RSCP(P): %d | SNR(P): %d | Tx Power: %d | BLER (Num Blocks
2606666601:navrf_chipset.c:Unable to read Device Chip ID, using non-AU default for GPIO
```

```
D0A6C738 loc_D0A6C738:                          ; CODE XREF: lte_rrc_dispatcher_loop+4C↑j
D0A6C738                { call lte_rrc_init_default_hdr
D0A6C73C                  immext
D0A6C740                  R1 = 0x40D1410 ; R0 = memw (R19 + 4) } ; LTE_RRC_PENDING_MSG_INDI
D0A6C744                { R21 = 0x17B8 }
D0A6C748                { R18 = memw (R17 + R21 << 0)
D0A6C74C                  if (cmp.gtu (R20, r18.new)) jump:t loc_D0A6C764 }
D0A6C750                { R1 = 0x17B0
D0A6C754                  immext
D0A6C758                  R0 = 0xC1C551EC ; R2 = R18 } ; lte_rrc_dispatcher.c:UTILS: memscpy, dst_size %d bytes < src_size %d, bytes
D0A6C75C                { call log_message_ }
D0A6C760                { R18 = memw (R17 + R21 << 0) }
```

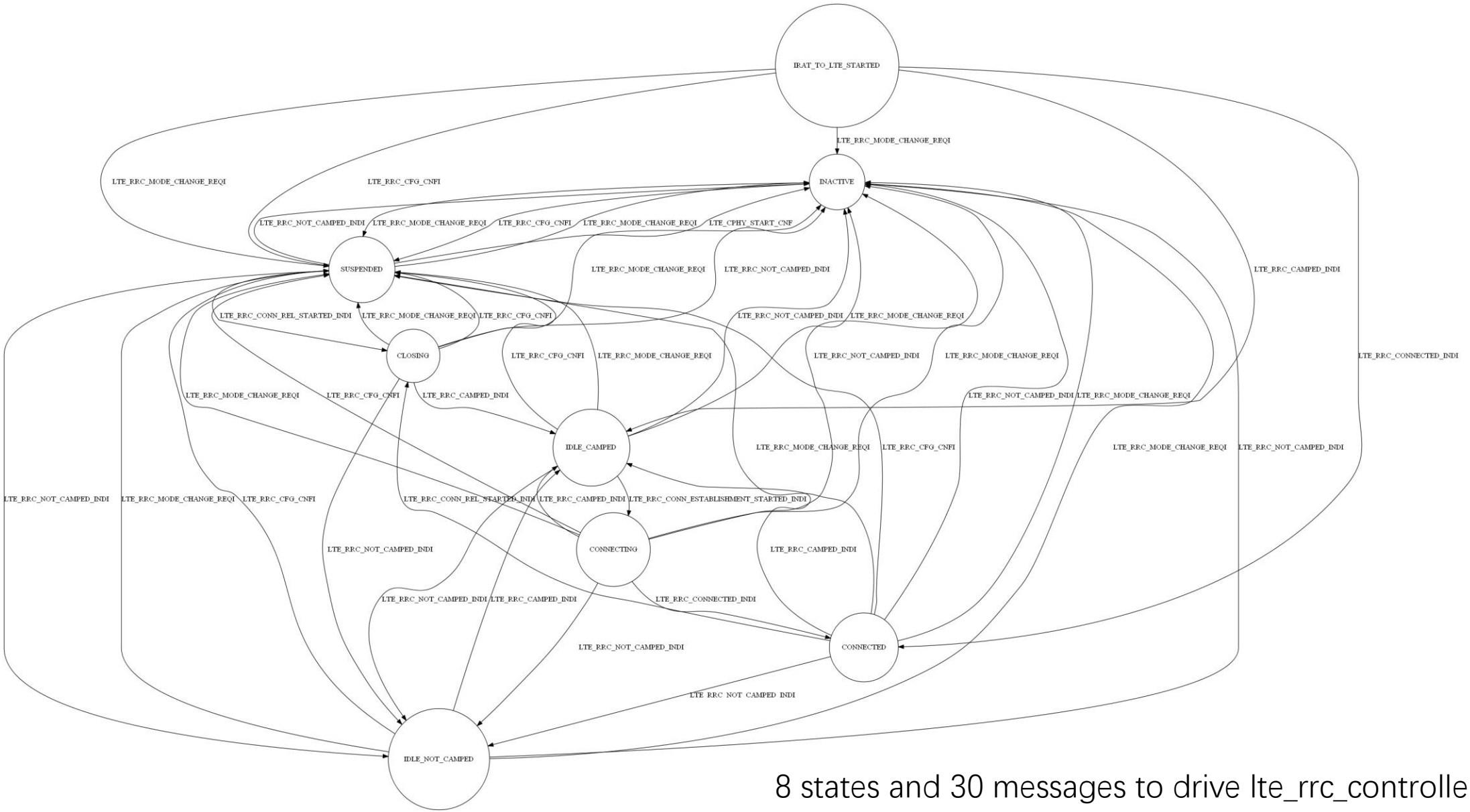
State Machine in LTE

- State machine and message
  - Functions: stm2_*, msgr_*
- stm2 (largely used to handle states transfer in rrc)
  - Initialized at function 0xD0A6BE34(ver.012511)
  - Beautiful structured in modem binary, including string to identify state and how to transfer from states
- msgr (message router)
  - UMID: 32bit uint value, Technology(1 byte) + Module(1 byte) + Type(1 byte) + ID(1 byte)
  - Broadcast based, N(sender) to M(receiver) connected by UMID
  - A good way to tracing message sender and receiver

```
:D0F4D738 48 D7 F4 D0 LTE_RRC_CONTROLLER_SM_preinst_constdata:dd LTE_RRC_CONTROLLER_SM
:D0F4D738                                                      ; DATA XREF: seg020:LTE_
:D0F4D73C 74 D7 F4 D0                dd LTE_RRC_CONTROLLER_SM_name ; "LTE_RRC_CONTROL
:D0F4D740 1A 2D 62 6F                dd 0x6F622D1A
:D0F4D744 00 00 00 00                dd 0
:D0F4D748 01 00 00 00 LTE_RRC_CONTROLLER_SM_constdata:dd 1    ; DATA XREF: seg020:LTE_
:D0F4D74C 08 00 00 00                dd 8
:D0F4D750 90 D7 F4 D0                dd LTE_RRC_CONTROLLER_SM_state_map
:D0F4D754 1E 00 00 00                dd 0x1E
:D0F4D758 10 D8 F4 D0                dd LTE_RRC_CONTROLLER_SM_input_map
:D0F4D75C 00 D9 F4 D0                dd LTE_RRC_CONTROLLER_SM_trans_map
:D0F4D760 38 30 BC C0                dd LTE_RRC_CONTROLLER_SM_entry_func
:D0F4D764 00 00 00 00                dd 0
:D0F4D768 6C F5 BE C0                dd lte_rrc_csp_sm_error
:D0F4D76C 74 F6 BE C0                dd LTE_RRC_CONTROLLER_SM_debug_hook
:D0F4D770 00 00 00 00                dd 0
:D0F4D774 4C 54 45 5F+LTE_RRC_CONTROLLER_SM_name:db "LTE_RRC_CONTROLLER_SM"
:D0F4D774 52 52 43 5F+                                         ; DATA XREF: seg020:D0F
:D0F4D774 43 4F 4E 54+               db 0
:D0F4D78A 00                         db    0
:D0F4D78B C0                         db 0xC0 ;
:D0F4D78C 00                         db    0
:D0F4D78D 7F                         db 0x7F ; ■
:D0F4D78E 00                         db    0
:D0F4D78F C0                         db 0xC0 ;
:D0F4D790 8A 31 E8 D0 LTE_RRC_CONTROLLER_SM_state_map:dd aInactive_0
:D0F4D790                                                      ; DATA XREF: seg020:D0F
:D0F4D790                                                      ; "INACTIVE"
:D0F4D794 C4 30 BC C0                dd 0xC0BC30C4
:D0F4D798 00 00 00 00                dd 0
:D0F4D79C 00 00 00 00                dd 0
:D0F4D7A0 05 DF F4 D0                dd aIdle_not_camped     ; "IDLE_NOT_CAMPED"
:D0F4D7A4 1C 57 BC C0                dd loc_C0BC571C
:D0F4D7A8 00 00 00 00                dd 0
:D0F4D7AC 00 00 00 00                dd 0
:D0F4D7B0 15 DF F4 D0                dd unk_D0F4DF15
:D0F4D7B4 60 59 BC C0                dd 0xC0BC5960
:D0F4D7B8 00 00 00 00                dd 0
:D0F4D7BC 00 00 00 00                dd 0
:D0F4D7C0 F7 74 E7 D0                dd aConnecting          ; "CONNECTING"
:D0F4D7C4 FC 59 BC C0                dd 0xC0BC59FC
:D0F4D7C8 00 00 00 00                dd 0
:D0F4D7CC 00 00 00 00                dd 0
:D0F4D7D0 B0 C5 F8 D0                dd aConnected           ; "CONNECTED"
:D0F4D7D4 84 5A BC C0                dd loc_C0BC5A84
:D0F4D7D8 00 00 00 00                dd 0
:D0F4D7DC 00 00 00 00                dd 0
:D0F4D7E0 F6 CA F6 D0                dd aSuspended           ; "SUSPENDED"
:D0F4D7E4 D0 5B BC C0                dd 0xC0BC5BD0
:D0F4D7E8 00 00 00 00                dd 0
:D0F4D7EC 00 00 00 00                dd 0
:D0F4D7F0 21 DF F4 D0                dd aIrat_to_lte_started ; "IRAT_TO_LTE_STARTED"
:D0F4D7F4 A8 5F BC C0                dd loc_C0BC5FA8
```

```
C0921A04 02 CA 00 54                { call sub_C0922B68 }
C0921A08 28 70 40 00                { immext
C0921A0C 82 40 00 78                  R2 = 0x40F0804
C0921A10 E1 41 02 78                  R1 = 0x40F
C0921A14 00 EB 1D B0                  R0 = add (R29, 0x158) }
C0921A18 84 F8 A6 5B                { call msgr_init_hdr_all_ }
C0921A1C 02 6B 1D B0                { R2 = add (R29, 0x158)
C0921A20 00 6B 1D B0                  R0 = add (R29, 0x158)
C0921A24 01 03 00 78                  R1 = 0x18 }
C0921A28 FC 79 A6 5B                { call msgr_send        ; errno_enum_type msgr_send
C0921A28                                                    ; (
C0921A28                                                    ;   /*! Pointer to message to be sent. Note that the first member of the message
C0921A28                                                    ;    *  that is pointed to by this pointer must be of msgr_hdr_struct_type type
C0921A28                                                    ;    */
C0921A28                                                    ;   msgr_hdr_struct_type      *msg_ptr,
C0921A28                                                    ;   /*! Total message length (header and payload) in bytes */
C0921A28                                                    ;   uint32                     msg_len
C0921A28                                                    ; );
C0921A2C 50 4E 44 0C                  immext
C0921A30 24 30 2C 69                  R20 = 0xC4439412 ; memw (R2 + 0x10) = #0 }
```

State Machine of LTE RRC



8 states and 30 messages to drive lte_rrc_controller run

Agenda

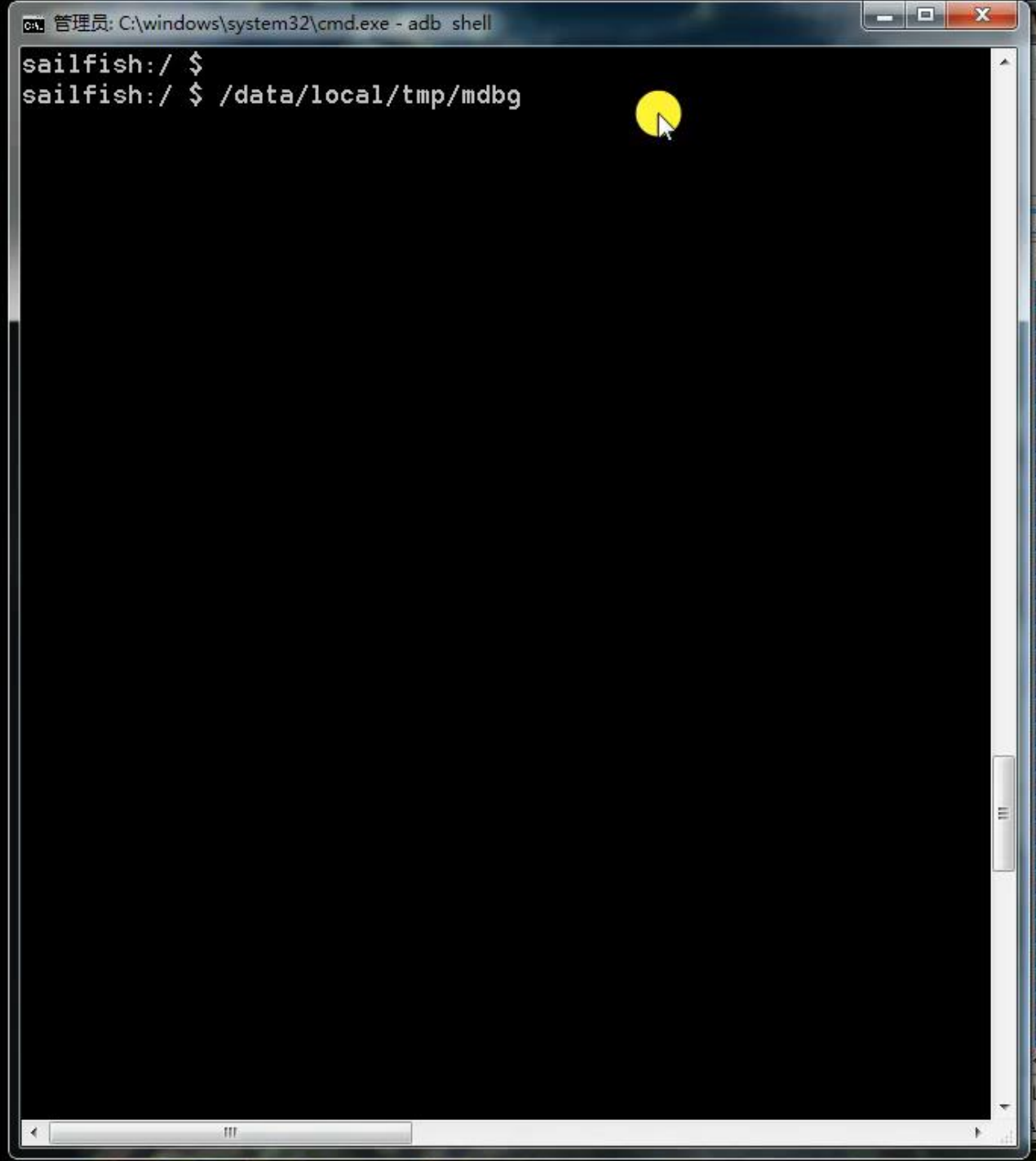
- Enter Qualcomm Modem World
- Static Analysis of Modem
- **Debugging Modem with ModKit**
- LTE Attack Surface Introduction

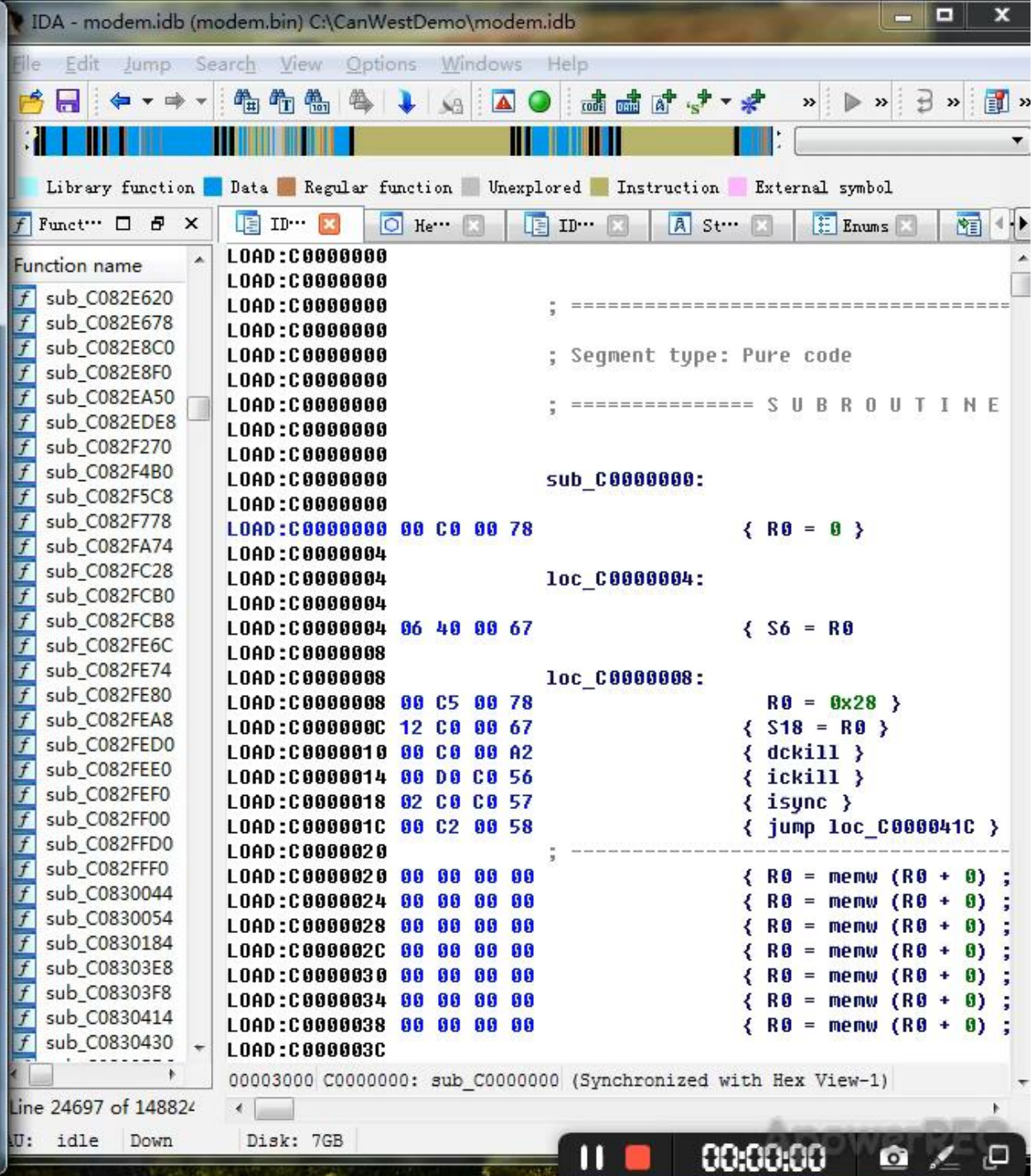
Modem Live Debugging

- Needs develop board and hardware debugger
  - Expensive (about 10k $?)
  - Can’t debug released product like Google Pixel
- Qualcomm Secure Boot disallow modify modem image
  - MBA (Modem Boot Authenticator)
- A bug can bypass the MBA to inject Hexagon code
  - Ability to read/write modem memory at any time from the Linux kernel
  - Reported to Qualcomm, patch in development, currently under embargo
- ModKit
  - A tool can be used as command executor on modem side
  - Debug server and in memory fuzzer

ModKit Debug Functions (Google Pixel)

- Primitive
  - Read/Write Modem memory at any time from Linux kernel
- Setup software breakpoints on modem execution
  - Read / Write Memory
  - Dump Registers
  - Dump Backtrace
- Setup condition for a breakpoint
  - Memory, Registers, Immediate Value

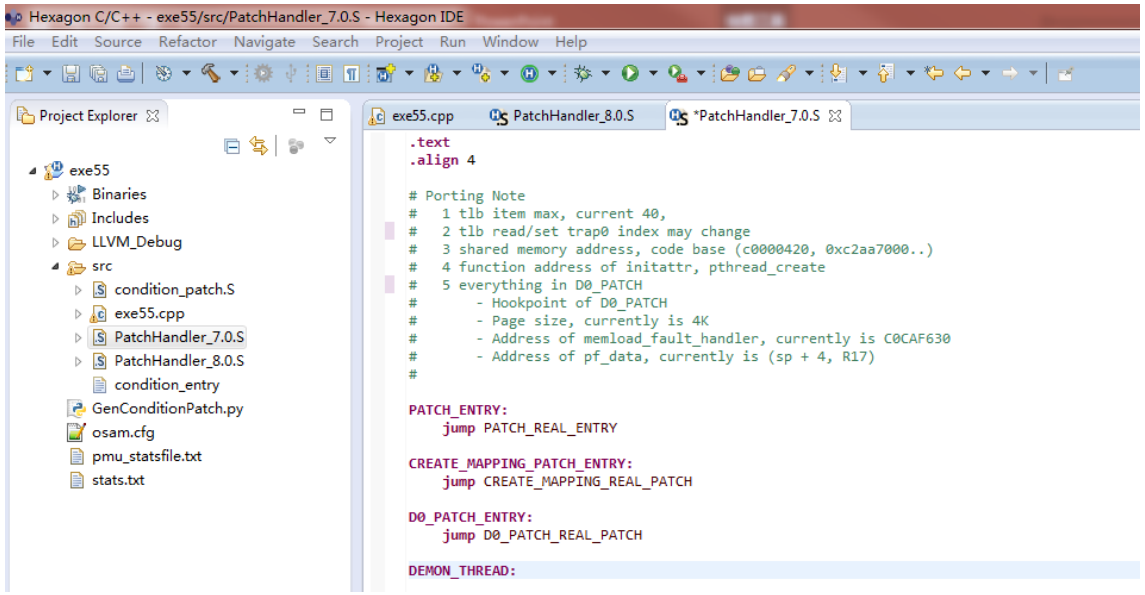

管理员: C:\windows\system32\cmd.exe - adb shell
sailfish:/ $
sailfish:/ $ /data/local/tmp/mdbg


IDA - modem.idb (modem.bin) C:\CanWestDemo\modem.idb
File Edit Jump Search View Options Windows Help
Library function Data Regular function Unexplored Instruction External symbol
Function name
sub_C082E620
sub_C082E678
sub_C082E8C0
sub_C082E8F0
sub_C082EA50
sub_C082EDE8
sub_C082F270
sub_C082F4B0
sub_C082F5C8
sub_C082F778
sub_C082FA74
sub_C082FC28
sub_C082FCB0
sub_C082FCB8
sub_C082FE6C
sub_C082FE74
sub_C082FE80
sub_C082FEA8
sub_C082FED0
sub_C082FEE0
sub_C082FEF0
sub_C082FF00
sub_C082FFD0
sub_C082FFF0
sub_C0830044
sub_C0830054
sub_C0830184
sub_C08303E8
sub_C08303F8
sub_C0830414
sub_C0830430
Line 24697 of 148824
AU: idle Down Disk: 7GB
LOAD:C0000000
LOAD:C0000000 ; ========== S U B R O U T I N E
LOAD:C0000000 sub_C0000000:
LOAD:C0000000 00 C0 00 78 { R0 = 0 }
LOAD:C0000004 loc_C0000004:
LOAD:C0000004 06 40 00 67 { S6 = R0
LOAD:C0000008 loc_C0000008:
LOAD:C0000008 00 C5 00 78 R0 = 0x28 }
LOAD:C000000C 12 C0 00 67 { S18 = R0 }
LOAD:C0000010 00 C0 00 A2 { dckill }
LOAD:C0000014 00 D0 C0 56 { ickill }
LOAD:C0000018 02 C0 C0 57 { isync }
LOAD:C000001C 00 C2 00 58 { jump loc_C000041C }
LOAD:C0000020 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C0000024 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C0000028 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C000002C 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C0000030 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C0000034 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C0000038 00 00 00 00 { R0 = memw (R0 + 0) ;
LOAD:C000003C
00003000 C0000000: sub_C0000000 (Synchronized with Hex View-1)

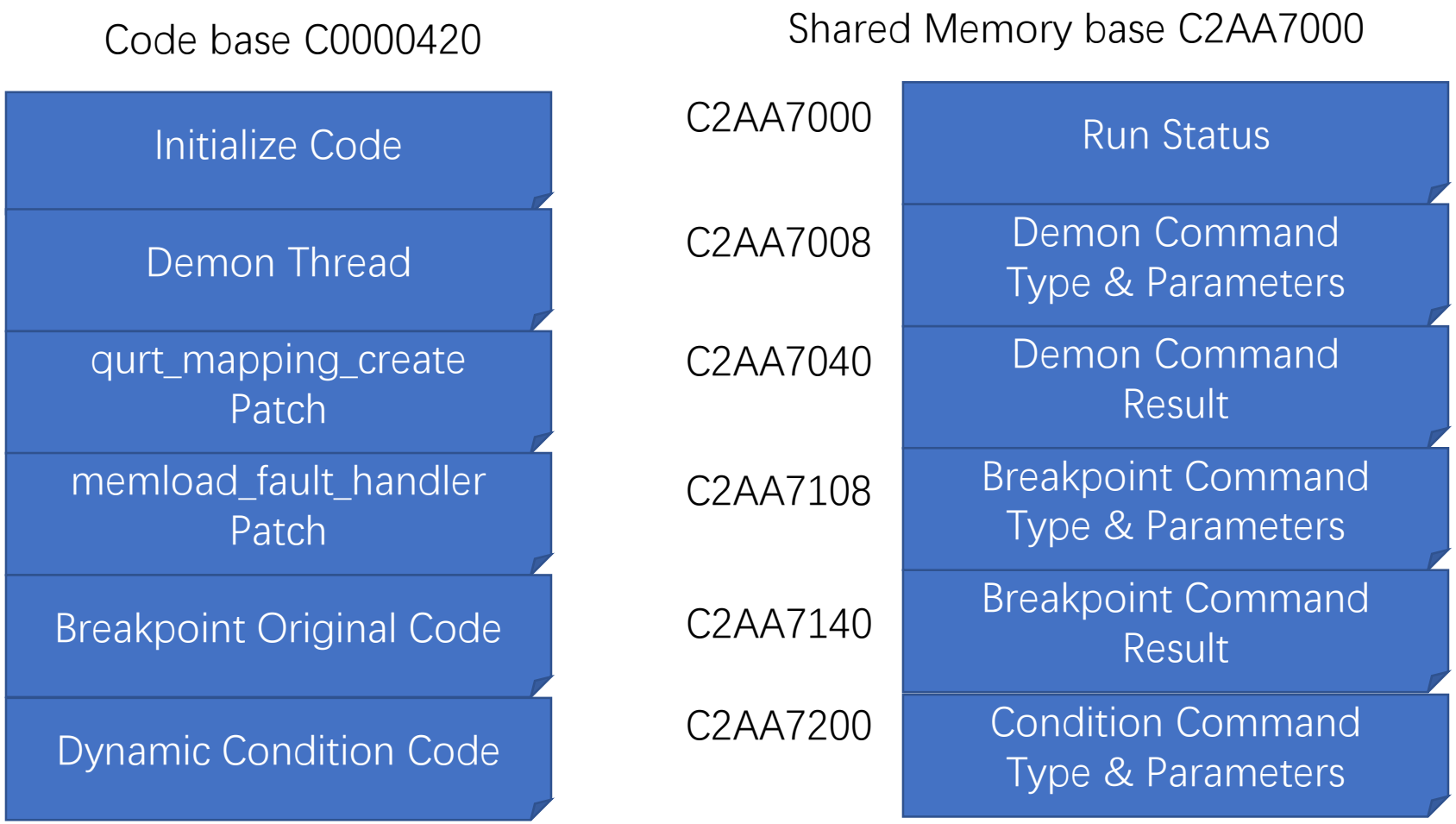
00:00:00

Prepare Debug Server Code

- Write debug server using Hexagon ASM
- Compile debug server in Hexagon SDK
- Extract debug server binary from .o



Debug Server-Memory Layout



Debug Server-Memory Layout

Debug Server Code Base

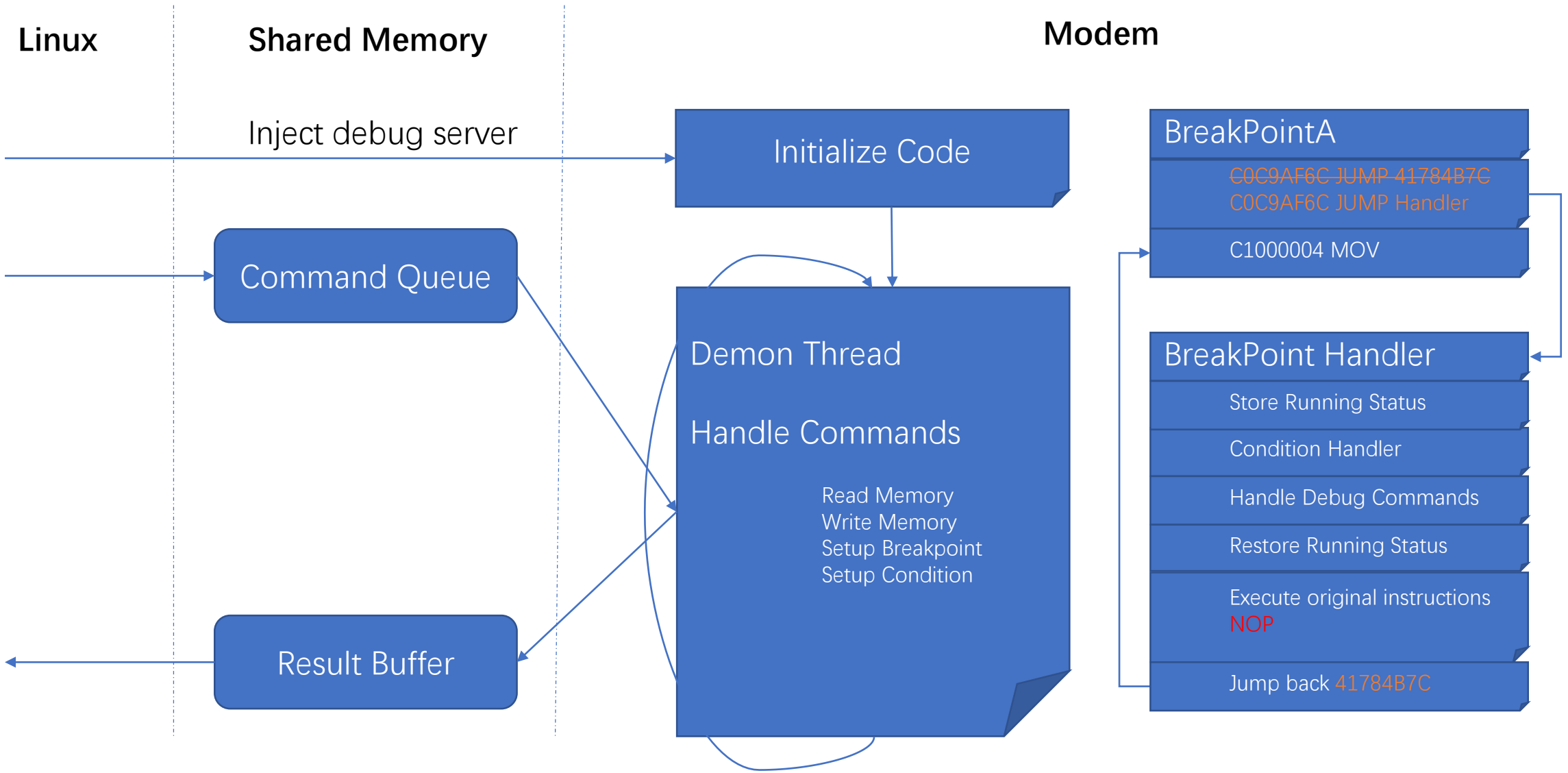
Shared Memory Base

LOAD	C0000000	C0001554	R	.	X	.	L	mempage	0001	public	CODE	32	0013
LOAD	C0010000	C0040000	R	W	.	.	L	mempage	0002	public	DATA	32	0013
LOAD	C0040000	C02EFAD4	R	.	X	.	L	mempage	0003	public	CODE	32	0013
LOAD	C0300000	C0622ACB	R	.	X	.	L	mempage	0004	public	CODE	32	0013
LOAD	C0640000	C0667DD0	R	W	X	.	L	mempage	0005	public	CODE	32	0013
LOAD	C0668000	C071BAC0	R	.	X	.	L	mempage	0006	public	CODE	32	0013
LOAD	C0720000	C091C064	R	.	X	.	L	mempage	0007	public	CODE	32	0013
LOAD	C091D000	C17BEE30	R	.	X	.	L	mempage	0008	public	CODE	32	0013
LOAD	C17C0000	C1813E60	R	W	.	.	L	mempage	0009	public	DATA	32	0013
LOAD	C1840000	C18B7400	R	W	.	.	L	mempage	000A	public	DATA	32	0013
LOAD	C18C0000	C23BD928	R	.	.	.	L	mempage	000B	public	DATA	32	0013
LOAD	C23C0000	C2AA62E0	R	W	.	.	L	mempage	000C	public	DATA	32	0013
LOAD	C2AA7000	C43F30C0	R	W	.	.	L	mempage	000D	public	BSS	32	0013
LOAD	C43F4000	C44081C0	R	W	.	.	L	mempage	000E	public	DATA	32	0013
LOAD	C4409000	C448A7F9	R	.	.	.	L	mempage	000F	public	DATA	32	0013
LOAD	C448B000	C4DDC000	R	.	.	.	L	mempage	0010	public	DATA	32	0013
LOAD	C4DDD000	C4DF0000	R	W	.	.	L	mempage	0011	public	DATA	32	0013
LOAD	C4DF1000	C4F34000	R	W	X	.	L	mempage	0012	public	CODE	32	0013
LOAD	C4F35000	C4F8EE54	R	W	.	.	L	mempage	0013	public	DATA	32	0013
LOAD	C4F8F000	C61F0000	R	W	.	.	L	mempage	0016	public	BSS	32	0000

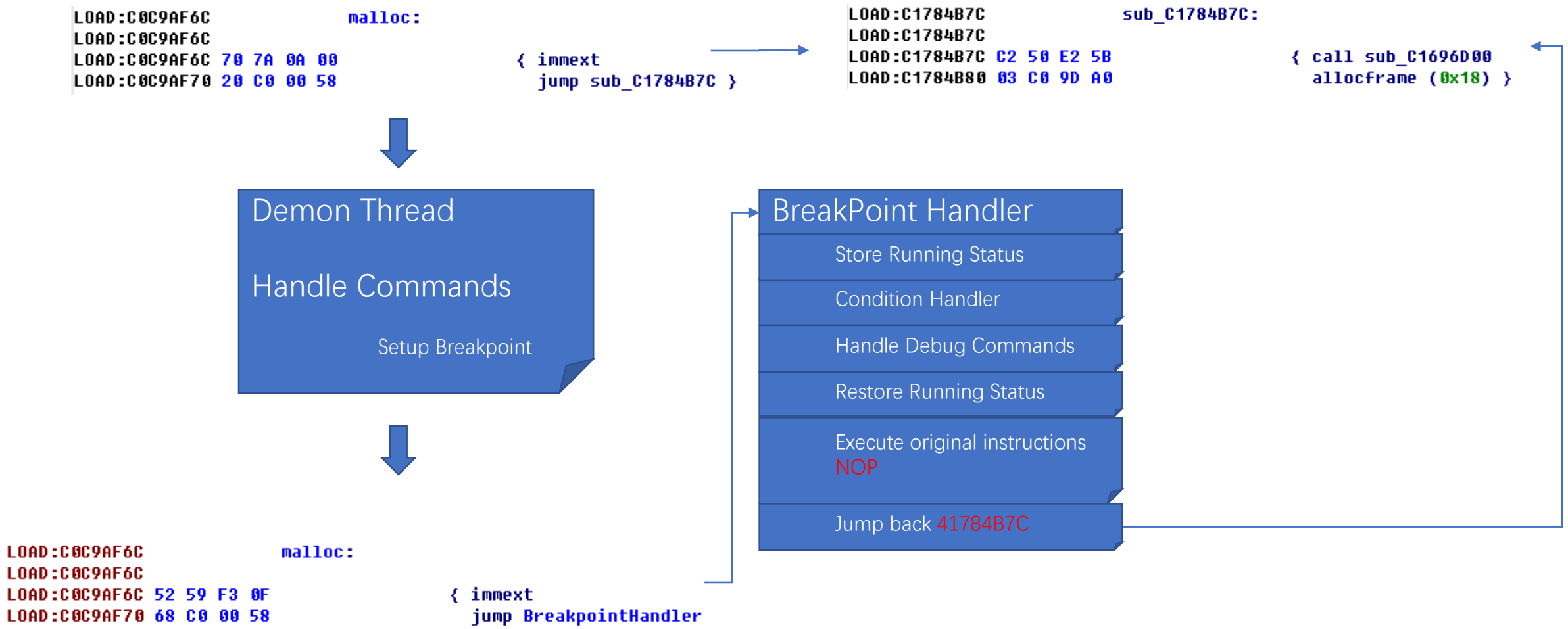
Debug Server Component

- Demon Thread
  - An infinitely loop running on Modem
  - Handle debug command
    - Read/Write memory immediately
    - Setup breakpoint
    - Setup breakpoint condition
- Breakpoint Handler
  - The injected code at the breakpoint
  - Handle debug command when hit a breakpoint
    - Read/Write memory
    - Dump registers/backtrace
- Condition Handler
  - The injected code at the breakpoint
  - Handle condition command when hit a breakpoint

Debug Server Implementation

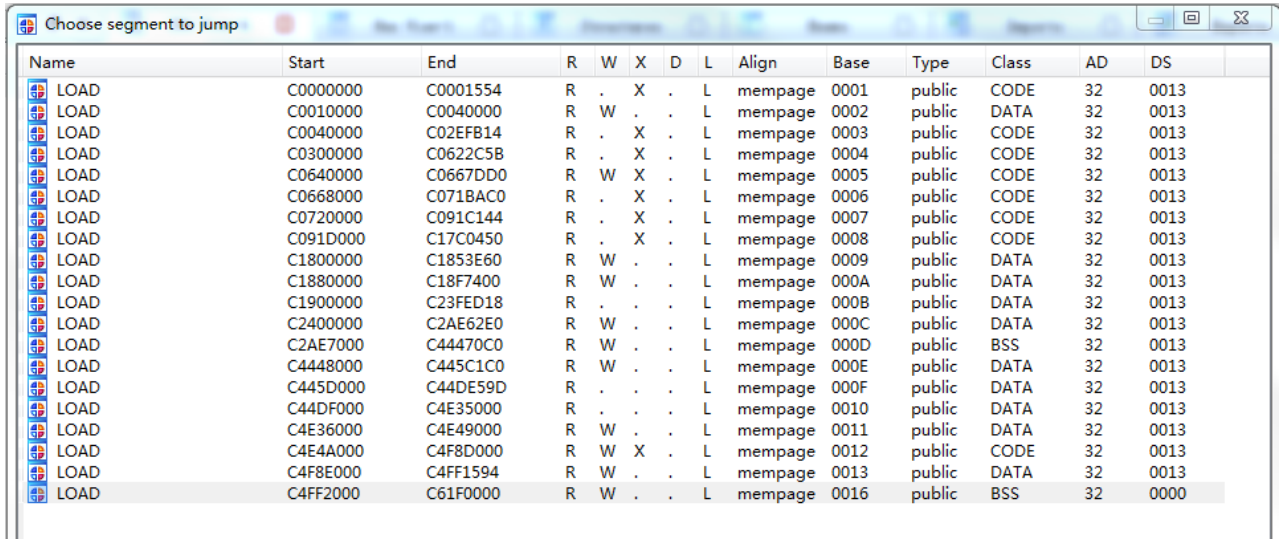


Breakpoint Implementation



Trouble Shooting-0xD0000000

```
LOAD:C0BB94F0                   sub_C0BB94F0:
LOAD:C0BB94F0 85 64 FE 00                       { immext
LOAD:C0BB94F4 78 C0 00 58                         jump loc_D0A4B66C }
LOAD:C0BB94F4                   ; End of function sub_C0BB94F0
```



Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	DS
LOAD	C0000000	C0001554	R	.	X	.	L	mempage	0001	public	CODE	32	0013
LOAD	C0010000	C0040000	R	W	.	.	L	mempage	0002	public	DATA	32	0013
LOAD	C0040000	C02EFB14	R	.	X	.	L	mempage	0003	public	CODE	32	0013
LOAD	C0300000	C0622C5B	R	.	X	.	L	mempage	0004	public	CODE	32	0013
LOAD	C0640000	C0667DD0	R	W	X	.	L	mempage	0005	public	CODE	32	0013
LOAD	C0668000	C071BAC0	R	.	X	.	L	mempage	0006	public	CODE	32	0013
LOAD	C0720000	C091C144	R	.	X	.	L	mempage	0007	public	CODE	32	0013
LOAD	C091D000	C17C0450	R	.	X	.	L	mempage	0008	public	CODE	32	0013
LOAD	C1800000	C1853E60	R	W	.	.	L	mempage	0009	public	DATA	32	0013
LOAD	C1880000	C18F7400	R	W	.	.	L	mempage	000A	public	DATA	32	0013
LOAD	C1900000	C23FED18	R	.	.	.	L	mempage	000B	public	DATA	32	0013
LOAD	C2400000	C2AE62E0	R	W	.	.	L	mempage	000C	public	DATA	32	0013
LOAD	C2AE7000	C44470C0	R	W	.	.	L	mempage	000D	public	BSS	32	0013
LOAD	C4448000	C445C1C0	R	W	.	.	L	mempage	000E	public	DATA	32	0013
LOAD	C445D000	C44DE59D	R	.	.	.	L	mempage	000F	public	DATA	32	0013
LOAD	C44DF000	C4E35000	R	.	.	.	L	mempage	0010	public	DATA	32	0013
LOAD	C4E36000	C4E49000	R	W	.	.	L	mempage	0011	public	DATA	32	0013
LOAD	C4E4A000	C4F8D000	R	W	X	.	L	mempage	0012	public	CODE	32	0013
LOAD	C4F8E000	C4FF1594	R	W	.	.	L	mempage	0013	public	DATA	32	0013
LOAD	C4FF2000	C61F0000	R	W	.	.	L	mempage	0016	public	BSS	32	0000

- Where is the code of D0000000?
  - The code at D0000000 is compressed
  - Page table isn't setup for D0000000 by default
  - Visit D0000000 will cause a page fault exception
  - The mem_load_exception will catch and fix it

Trouble Shooting-0xD0000000

- So how to get the code of D0000000?
  - Simply read the memory out using ModKit
  - Of course you can unzip the code by yourself

- So how to setup breakpoint on D0000000?
  - That's what mem_load_handler Patch doing
  - Each time a new page fault exception occurs
    - Corresponding code is loaded into memory (by mem_load_handler)
    - Corresponding page table is setup (default by mem_load_handler)
    - And then the code is patched (by our patch)
  - There is a page table cache (maybe LRU)
    - Should patch all the breakpoints every time
    - To avoid page reloading result to patch missing

System APIs Used

API Name	Usage	Address[1]
qurt_tlb_entry_read	Read original TLB info [2]	trap0(#0x45) [3]
qurt_tlb_entry_set	Modify TLB flags to RWX	trap0(#0x44) [3]
pthread_create	Create Demon Thread	C1758A60
pthread_attr_init	Init Demon Thread Attribute	C1758C20
qurt_mapping_create	Hook to modify mapping attribute to RWX	C173F3D4
memload_fault_handler	Hook to modify code of D0000000	C0CAF0E8

[1] Address of Android factory image sailfish-nde63h

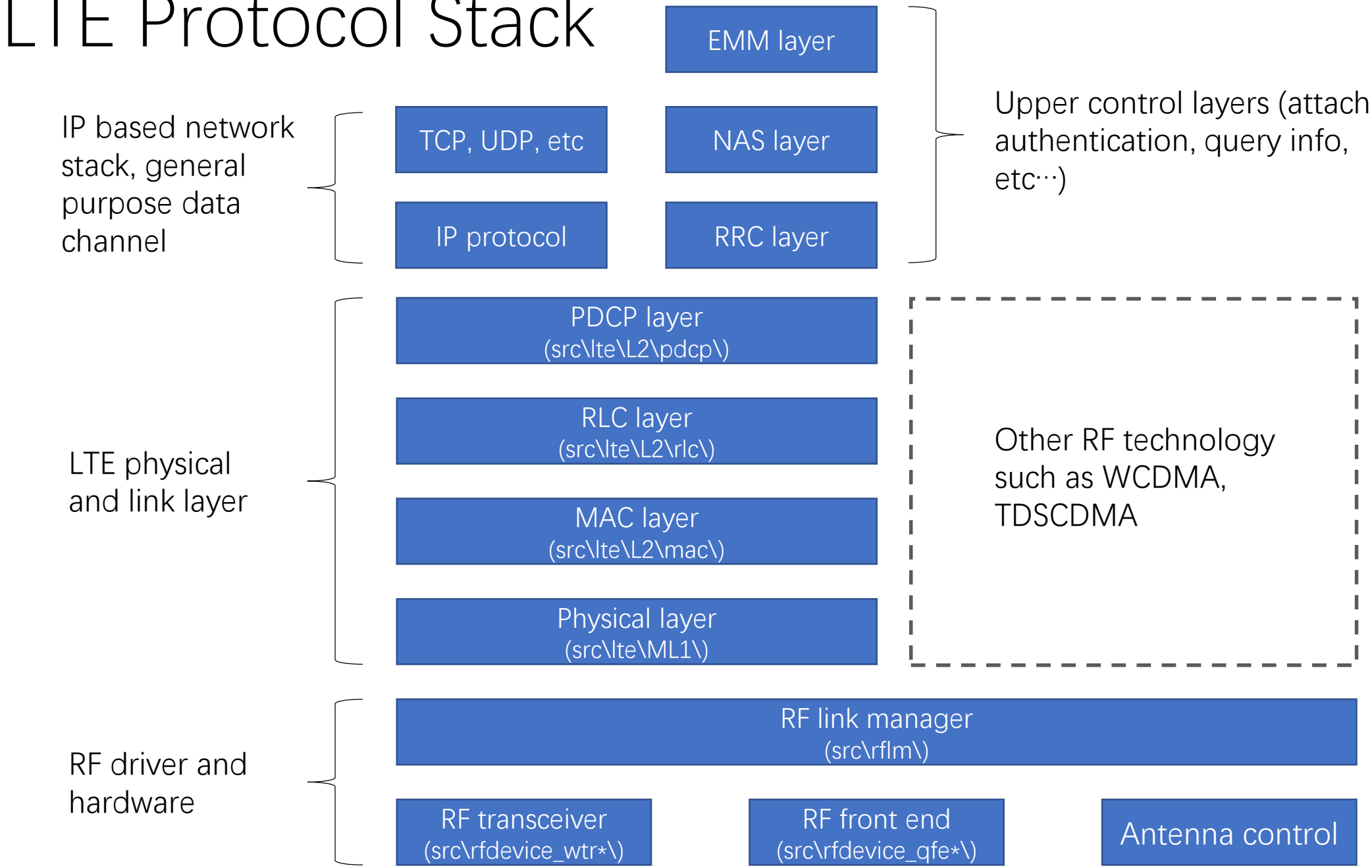
[2] TLB - Translation Lookaside Buffer

[3] The number may be different from versions. But the code sequence are similar.
You can search the code sequence to find the function.

Agenda

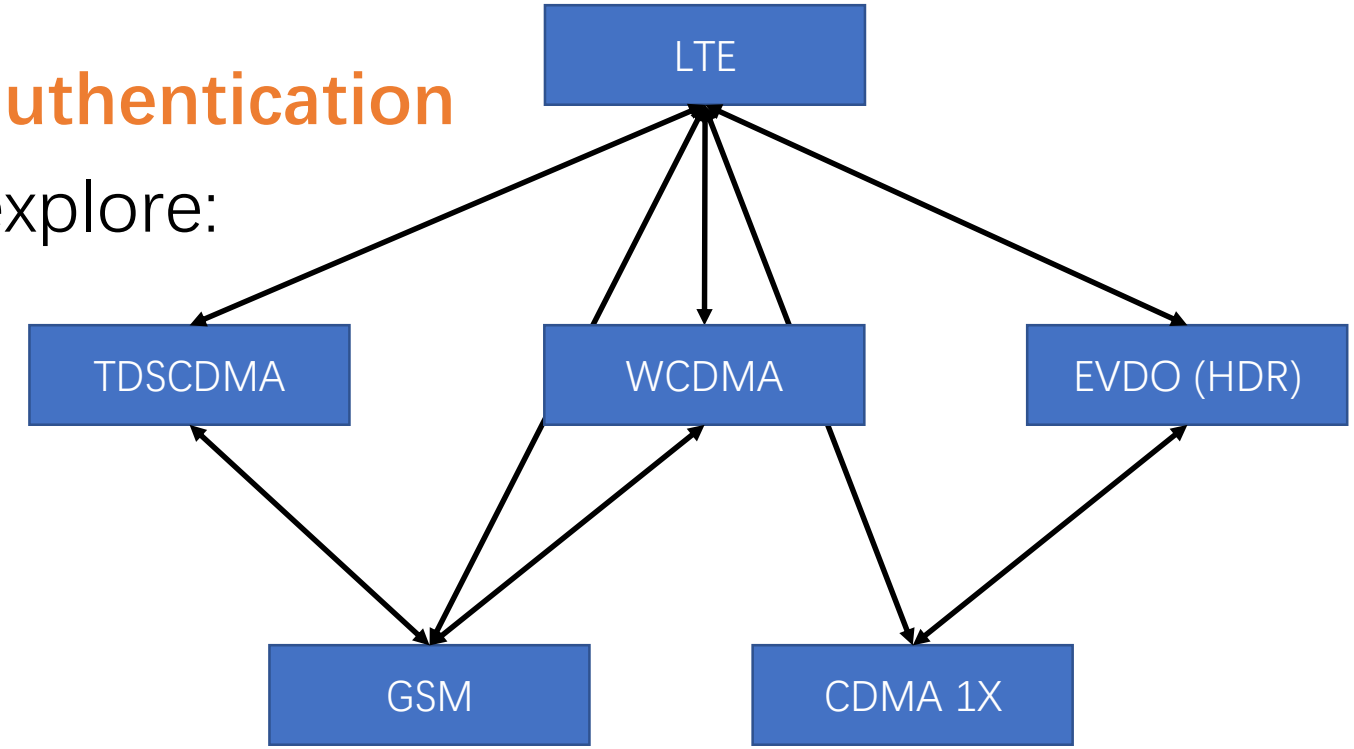
- Enter Qualcomm Modem World
- Static Analysis of Modem
- Debugging Modem with ModKit
- **LTE Attack Surface Introduction**

LTE Protocol Stack

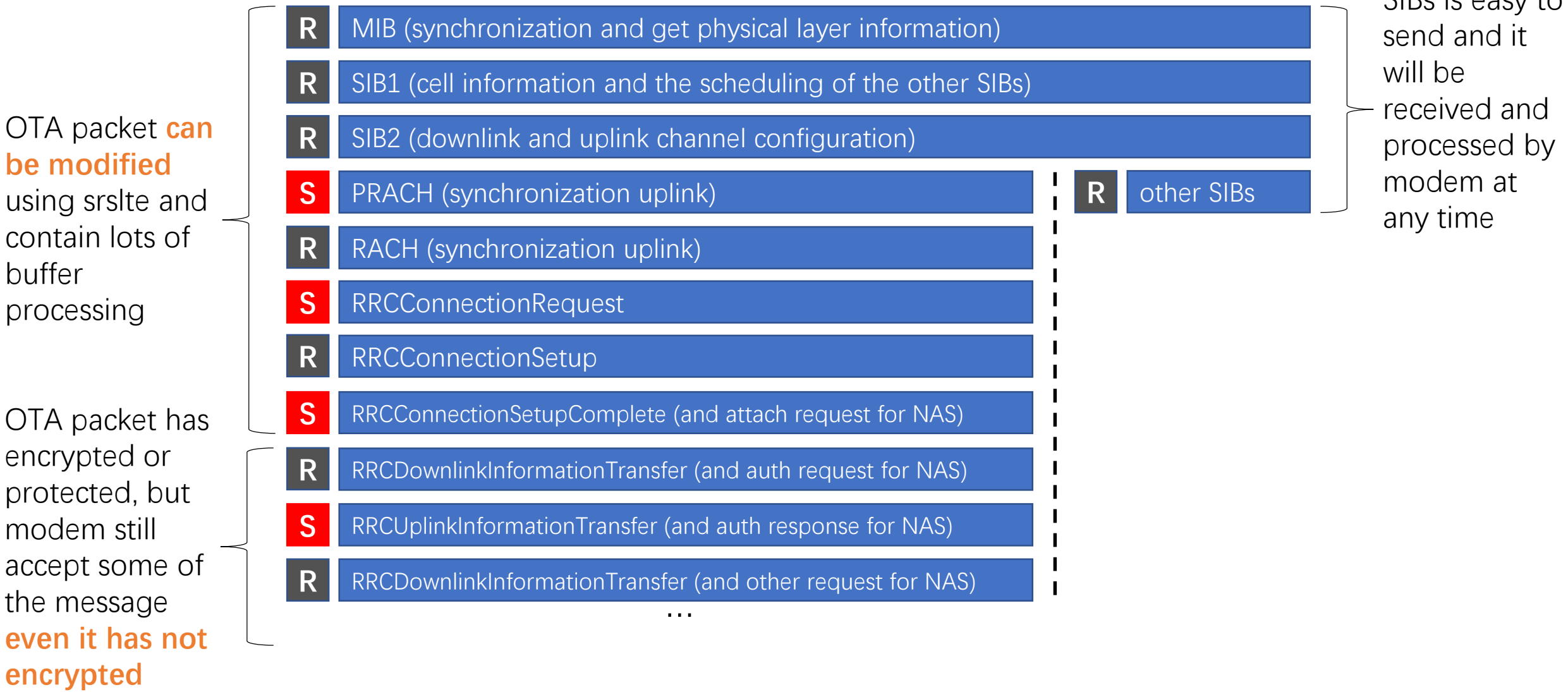


OTA attack surface analysis (1)

- LTE has no dedicated audio or video service domain
- To make a phone call, VoLTE or switch to older RF technology is needed
- **Most switch happens before authentication**
- Many RF switch technology to explore:
  - IRAT handover
  - Cell Redirection
  - CSFB



OTA attack surface analysis (2)


R MIB (synchronization and get physical layer information)
R SIB1 (cell information and the scheduling of the other SIBs)
R SIB2 (downlink and uplink channel configuration)
S PRACH (synchronization uplink)
R RACH (synchronization uplink)
S RRCConnectionRequest
R RRCConnectionSetup
S RRCConnectionSetupComplete (and attach request for NAS)
R RRCDownlinkInformationTransfer (and auth request for NAS)
S RRCUplinkInformationTransfer (and auth response for NAS)
R RRCDownlinkInformationTransfer (and other request for NAS)
...
R other SIBs
OTA packet can be modified using srslte and contain lots of buffer processing
OTA packet has encrypted or protected, but modem still accept some of the message even it has not encrypted
SIBs is easy to send and it will be received and processed by modem at any time

Exploit environment

- NO ASLR, and a lot of hardcoded magic address
  - such as Modem firmware will always load at physical address 0x88800000 and virtual address 0xC0000000 (Google Pixel)
- Memory Permission Protected
  - Code segment is not writable
  - Data segment is not executable (DEP)
- Stack Protection
  - Stack bounds protection, FRAMELIMIT register stores the lower bound
  - Stack canary to protect stack smashing, XORed with FRAMEKEY register
- Heap Protection
  - Heap management by QuRT
  - Each block has a header which is protected by magic number
  - Active and Freed blocks have different magic numbers

THANK YOU